2013 7th Asia Modelling Symposium

SST: An Efficient Suffix-Sharing Trie Structure for Dictionary Lookup

Komate Amphawan
Computational Innovation Laboratory
Faculty of Informatics, Burapha University
Chonburi, 20131, Thailand
komate@gmail.com

***Abstract*—In many information systems, a dictionary lookup is widely adopt and utilized as a part of computation. To use in-memory dictionary, one of the most popular techniques is the using of trie structure to store a collection of words. However, it is well-known that the using of trie will consume a large amount of memory when the collection of words is large and it cannot notify the nature of words which may help users to better understand context. Therefore, in this paper, an efficient Suffix-Sharing Trie structure, named *SST*, is proposed to store all of words. *SST* is a two-trie structure that not only share prefixes but it can also share suffixes of words. By doing this, *SST* can identify which words having a suffix as a component and it can also reduce the memory consumption from the trie structure. Experimental results show that the proposed *SST* structure is efficient on both runtime and memory usage to maintain a collection of words in-memory.**

Keywords-component; dictionary lookup; trie structure; suffix sharing

I. INTRODUCTION

In the era of information technology, natural language processing (*NLP*) plays a significant role for learning and understanding the human communication methods and contexts. In general, NLP concerns with machine translation [1], automatic summarization [2] [3], morphological segmentation [4] [5], named-entity recognition [6] and so on. From the above applications, there are several tasks applying dictionary as a part of computation, for example, searching on vocabulary, grammar checking, word segmentation, text compression, etc.

In the past, it is well-known that dictionary creation is labored and time consuming. This attracts many researchers to make this process automatically. To construct dictionary, there are several well-known techniques and improvements that have different advantages and disadvantages, for instance, rank and select [7], using hashing function, etc. However, there is another widely used technique that is the using of trie structure to maintain vocabulary due to it is simple, easy to understand and it has also a good performance. Hence, tries are extensively applied for storing and matching strings over a given alphabet. There are several applications using of tries include dictionary lookup for text processing [8] [9], itemset lookup for mining association rules from retail transactional databases [10] [11] [12], IP address look-up in network routers [13] [14]. These algorithms aim to store and collect strings in memory. However, they did not consider about extracting information from a given collection of strings.

Therefore, this paper aims to improve the performance of the trie structure by developing a new structure to collect and maintain a collection of strings (words) in-memory where it can provide some information. Then, an efficient suffix-sharing trie structure, also called *SST*, is presented. The main advantage of SST is not only share prefixes like the normal property of tree and trie structure but it can also share suffixes of words which helps users to identify which words having suffix as a component. Moreover, this may help to better under-stand structure and context of sentences. With the suffix-sharing property, *SST* can also reduce memory usage to collect all words in-memory. From the experimental studies, it can be seen that the proposed *SST* structure can efficiently maintain a set of words (in-memory dictionary) in both runtime and memory consumption.

The organization of the rest of this paper is as follows. Section 2 mentions on basics concept and related data structures. Section 3 introduces a new efficient Suffix-Sharing Trie structure. Experimental results are shown in section 4. Section 5 gives you a conclusion of this paper.

II. BASIC CONCEPTS AND RELATED DATA STRUCTURES

In general, there are several data structures for capturing vocabulary in-memory including hash-tables, search tree, tries, etc. Hence, this section will give you a brief review of these structures and the definition of suffix that will be used in this paper.

A. Hash Tables

A hash table collects all records by using a hash function to map a word into a bucket. The naïve hashing is done by linear probing; however, it is not efficient enough. Then, it attracts a lot of intension from researchers to improve hashing functions [15] [16]. However, there are two hashing functions that are very efficient and commonly used, namely, *Bitwise* and *Perfect hashing*.

1) Bitwise hashing

As proposed in [17], *Bitwise hashing* used a non-prime number as its seed. The advantage of this method is that the bucket size is the power of 2. Typically, most of the hashing

978-0-7695-5101-2/13 $26.00 © 2013 IEEE
DOI 10.1109/AMS.2013.33

functions need the complex mathematical operations while bitwise hashing function adopts only shift bit operations. By doing this, it is claimed that their proposed method is 10–30 times faster than the naïve hashing.

2) Perfect hashing

The main advantage of perfect hashing is that no collision appears in the insertion and lookup process. Under this approach, each word is assigned to a unique hashing integer which causes the look-up time is guaranteed to be $O(1)$. However, the major disadvantage of perfect hashing is that the bucket size is the number of all possible words.

B. Search tree

A search tree is a tree structure that starts with a root node and then traverses to each node until leaf. One of the most popular search tree structures is the *Binary Search Tree* (*BST*) where every node's left sub-tree has keys less than the node's key and every node's right sub-tree has keys greater than node's key. The performance of the *BST* depends on the height of the tree. The average search time is $O(n \log n)$ but it takes $O(n)$ for the worst case. There are a lot of variations of BST such as: AVL, red-black and splay trees, that are widely applied in many applications.

C. Trie

Knuth [18] defines a trie structure as an n-ary tree whose nodes are *N-place vectors* with components corresponding to individual symbols in the alphabets. Each node on level x expresses the set of all strings that begins with certain sequence of x symbols; the node specifies an n-way branch depending on the $(x+1)^{th}$ symbol. To avoid confusion between keys like "an" and "ant", let us add a special symbol "#" at the end of all strings, so no prefix of a key can be a key itself.

Let K be a set of strings. In the trie structure, each path from the root to a leaf node corresponds to one string in K. Information for each string can be attached to the corresponding leaf (terminal) node uniquely because there exists a one-to-one correspondence between the strings and the terminal nodes. In the trie, common prefixes of strings can be shared. The number of transitions for rear strings extremely increases for a large set of keys, so it is important to compress transitions. Fig. 1 shows an example of trie structure that contains the words "an", "ant", "the", "then", and "than", respectively.

The searching for a string (word) in a trie structure starts from the root node and proceeds as a descent in the tree structure. At each level, out-links of node are examined to determine is all the next symbol in the given string appears as a label for one of the links. The search is successful when all the symbols in the strings are matched. On the other hand, it is unsuccessful if at any node, the next symbol in the string does not appear as a label for one of the outgoing links.

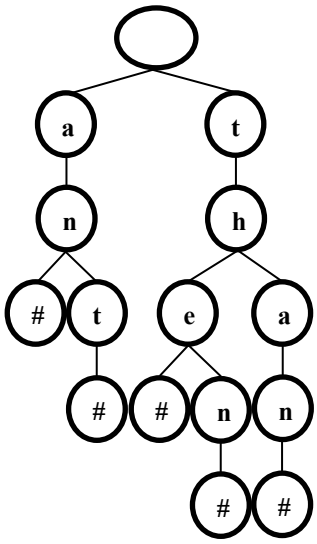

Figure 1. Example of trie structure containing five words

D. Suffix

A suffix is a letter or group of letters added to the end of a word or root (*i.e.*, a base form), serving to form a new word or functioning as an inflectional ending. There are two primary types of suffix in English: (*i*) a *derivational suffix* (such as the addition of "-ly" to an adjective to form an adverb) indicates what type of word it is, and (*ii*) an *inflectional suffix* (such as the addition of "-s" to a noun to form a plural) tells something about the word's grammatical behavior. Moreover, suffixes display all kinds of relationships between form, meaning, and function. Some are rare and have only vague meanings, as with the "-een" in "velveteen". Some have just enough uses to suggest a meaning, as with "-iff" in "bailiff", "plaintiff", suggesting someone involved with law. The number of suffixes in modern English is so great, and the forms of several, especially in words derived through the French from Latin, are so variable that an attempt to exhibit them all would tend to confusion.

III. THE PROPOSED SUFFIX-SHARING TRIE STURCTURE

Although there are many different data structures for words accumulation, few of them are efficient and aim to notify information of words. Then, this section introduces an efficient **S**uffix-**S**haring **T**rie structure (*SST*) for maintaining a collection of words. *SST* can identify words having suffix as a component, can alleviate the memory usage and may help users to find root of words. Under *SST* structure, there are two common-trie structures linked together that are (*i*) a *suffix-trie* containing only given suffixes of words with a unique running number at the end of each suffix, and (*ii*) a *word-trie* maintaining all words with the ability to share the same word-suffixes. By doing this, *SST* is not only share prefixes of words like a normal

property of the original tree and trie structure but it also has capability for sharing suffixes.

As shown on the left trie of Fig. 2, each node of the *suffix-trie* contains a letter and bi-directional links between two consecutive letters of suffixes (*i.e.* bi-directional links between parent and child nodes). From this trie, we can see that each suffix has a unique running number at the leaf node and all of them are stored in reverse manner where this can save time for comparing and identifying words having suffixes as a component. From the figure, there are five suffixes contained in *suffix-trie* that are (i) "able", (ii) "ible", (iii) "er", (iv) "or", and (v) "ment", respectively.

On the other hand, the right-hand side trie of Fig. 2 is the *word-trie* used for collecting all words. Each node of the *word-trie* contains one letter of words and links between two consecutive letters (*i.e.* links from parent to child node). Each path of *word-trie* is maintained in contrast to the previous mentioned *suffix-tire*. The *word-trie* stores all words in the common order, however, suffixes of words are eliminated by creating a link from *word-trie* to the *suffix-trie*. By doing this, *SST* can avoid to store suffixes of all words in the *word-trie*. This leads *SST* can reduce memory to hold words and has ability to identify words with suffixes. In Fig. 2, there are only one non-suffix word and five words with suffixes contained in the *word-trie*. For example, the words "enable" and "movable" have "able" as a suffix. Then, *SST* stores only non-suffix parts that are "en" and "mov" in the *word-trie* and then create a link from the end of each non-suffix to the leaf node (having a running number) of suffix "able" in the *suffix-trie*.

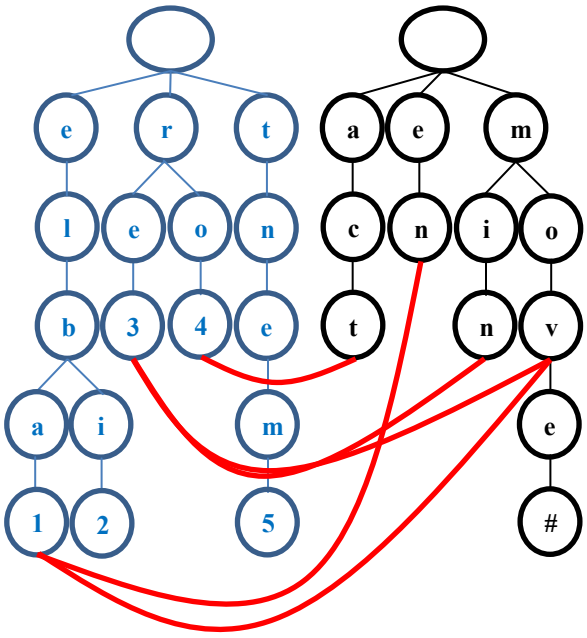

Figure 2. Example of SST structure

A. *SST construction*

The *SST* construction is composed of two steps: (*i*) suffix-trie construction and (*ii*) word-trie construction. To create suffix-trie, each suffix in the collection of suffixes is read and then insert into the suffix-trie in reverse manner for efficiently constructing and searching suffixes from *suffix-trie*. As illustrated in Fig. 2, all the suffixes are stored in the reverse manner. For example, the suffix "able" is maintained by "elba".

After gathering all suffixes, each word in the given collection of words is regarded by firstly checking whether it has suffix as a component or not. In the case that the considered word has suffix as a component, *SST* stores non-suffix part of the considered word in the word-trie and then create a link to a leaf node of the suffix-trie that has corresponding suffix as the considered word. On the other hand, all of the considered word is collected in the word-trie with a letter '#' at the end. For instance, the word "enable" is firstly examined to determine whether it has suffix as a component. In this case, we know that "enable" has "able" as a suffix. Thus, *SST* eliminates "able" out of "enable" and then collects the non-suffix part (*i.e.,* "en") to the word-trie and then create a link to the leaf node of suffix "able (elba)" in the *suffix-trie* at the leaf node of suffix "able (elba)". All the details of the *SST* construction are shown in Fig. 3.

B. *Searching words from SST*

Searching on the SST structure is a simple operation requiring traversal along the *suffix-trie* and *word-trie*. As shown the details in Fig. 4, *SST* starts to consider from the end of the given word in order to identify suffix of that word. To identify suffixes, the *suffix-trie* is traversed along the sibling node and if any letter matches, it traverses along the child nodes. After suffix identifying, *SST* knows that whether the given word has suffix as a component or not. Next, the word-trie is traversed by the non-suffix part of the given word.

To understand the searching procedure, please refer to Fig. 5 which is an example of searching word "enable" which has suffix "able" as a component. The searching starts from considering the last character that is 'e' and then traverses *suffix-trie* from the root node to find a child node of 'e'. After that the searching considers the second and other letters from the end to match with nodes in the trie. From the figure, we can see the dash-line in orange that is a path "e→l→b→a→1" with the running number '1' and the end of path matched characters "elba" of "enable". Then, we will recognize as the given word has a suffix with labeled "1". Next, the searching will take the non-suffix part ("en") to further search on the *word-trie*.

To search on the given non-suffix part, the first letter 'e' is considered by traversing *word-trie* from the root node to find a child node of 'e'. Fortunately, there is a node that matches 'e', then we go to that node and move the consideration to the second letter 'n'. For 'n', there is a chi-

SST Construction
Input: A collection of English words and suffixes
Output: The *SST* structure containing all words in memory

- initial roots of *suffix-trie* and *word-trie*
for each suffix *s* in a collection of suffix
  - initial *current-node* as the root of *suffix-trie*
  for each letter *l* started from the end of the suffix *s*
    - find a child node of the *current-node* that matches with letter *l*
    if there is a child node *n* matches with *l*
      - assign the *current-node* as the child node *n*
    else
      - create a new node *m* with letter *l* as a child node of the *current-node*
      - assign the *current-node* as the new node *m*

  - create a new node as a child node of the *current-node* with a unique running number

for each word *w* in a collection of English words
  - initial the *current-node* as the root of *suffix-trie*
  for each letter *l* started from the end of word *w*
    - find a child node of the *current-node* that matches with letter *l*
    if there is a child node *n* matches with *l*
      - assign the *current-node* as the child node *n*
    else
      if there is a child node *r* having a running number
        - remember the child node *r*
        - eliminate all the regarded letters out of *w*

    - terminate searching on the suffix-trie

- initial *current-node* as the root of *word-trie*
for each remaining letter *l* of word *w*
    - find a child node of the *current-node* that matches with letter *l*
    if there is a child node *n* matches with *l*
      - assign the *current-node* as the child node *n*
    else
      - create a new node *m* with letter *l* as a child node of the *current-node*
      - assign the *current-node* as the new node *m*

if word *w* has a suffix as a component
  - create a link from *current-node* to the previous remembered node in *suffix-trie*
else
  - create a new node with letter '#' as a child node of the *current-node*

Figure 3. Details of SST construction

Searching on SST

Input: a user-give word "*w*", The *SST* structure held in-memory
Output: "true" if SST found "*w*" in *SST* structure (also) return the running number at the leaf node in *suffix-trie* if *w* contains suffix), Otherwise, "false".

- initial *current-node* as the root of *suffix-trie*
for each letter *l* started from the end of a given word *w*
  - find a child node of the *current-node* that matches with letter *c*
  if there is a child node *n* matches with *l*
    - assign the *current-node* as the child node *n*
  else
    if there is a child node *r* having a running number
      - remember the found running number of the child node *r*
      - eliminate all the regarded letters out of word *w*

    - terminate searching on the *suffix-trie*

- initial the *current-node* as the root of word-trie
for each letter *l* of the word *w*
  - find a child node of the *current-node* that matches with letter *c*
  if there is a child node *n* matches with *l*
    - assign the *current-node* as the child node *n*
  else
    - return "false" (this means the word *w* is not a member of the given collection of words)

if there is a child node r having character "#"
  - return "true" (this means word *w* is a word in the given collection of words)
if there is a child node *r* having the same running number as the previous remembered
  - return "true with the running number" (this means word *w* is a word in the given collection of words and it has a suffix as a component)

Figure 4. Details of searching on SST structure

ld node of node 'n' of node 'e', we then move to that node. Since 'n' is the last letter of non-suffix part, we have to find a child node of 'n' that have the same running number as the remembered number that was recognized at the first step of searching. In this case, we found that both running number are the same, then we can conclude that the given word "enable" is a word in a collection of words and it has suffix as a component.

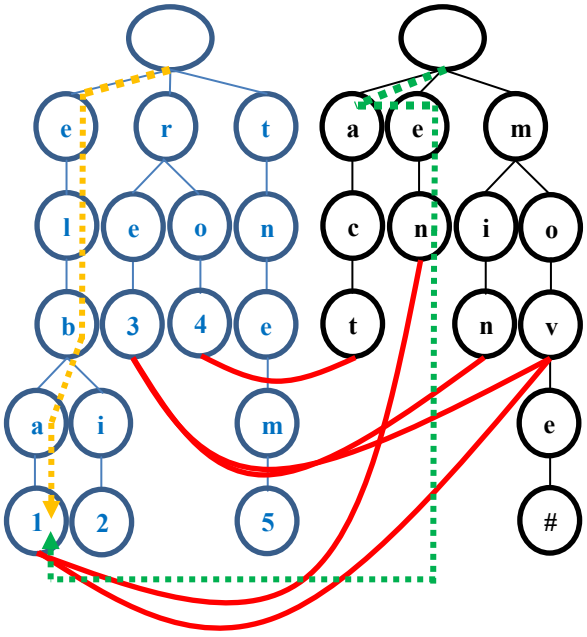

Figure 5. Example of searching the word "enable" on SST structure

IV. EXPERIMENTAL RESULTS

In this section, we evaluate the performance of the proposed *SST* to create dictionary by comparing with that of the *original trie-structure*. To investigate the performance, a collection of English words (there are two datasets: (*i*) a collection of English words collected from http://en.wiktionary.org/wiki/Index:English/, and (*ii*) a collection of suffixes gathered from http://en.wiktionary.org/wiki/Category:English_suffixes, separated in two files) is used. The input file contains 216,555 English words and 682 suffixes, respectively.

Experiments in this paper are conducted in order to observe performance in three aspects: (*i*) time for collecting words in memory, (*ii*) number of nodes and memory usage to hold all words, and (*iii*) time for searching words.

Fig. 6 shows runtime of the original trie structure and *SST* for storing all words in memory. From this figure, we can see that *SST* use more time than trie structure. This is because SST has to firstly construct suffix-trie and then create trie for all words with sharing suffix.

Fig. 7 and 8 show the number of nodes and memory consumption of the two data structures. SST can reduce the number of nodes from trie-structure since it can take advantage of suffix-sharing. Then, SST can decrease the memory usage up to 18.4%.

Fig. 9 and 10 show the runtime for searching words separated by number of letters of words. The runtime shown in the figures is the average time that searching for a group of words that have the same length. As we can see in the figures, SST spend more time than the using of trie structure to find words due to it has to firstly identify suffix for each word in the *suffix-trie* and then search that word in the *word-trie*. However, the gap of difference on searching time is not significant. The maximum different is only 2,500 ns which is very few and it is acceptable.

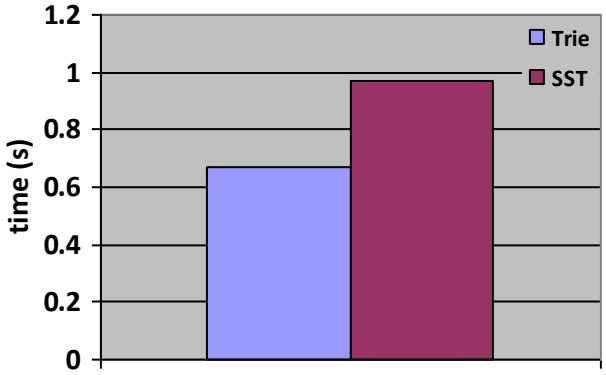

Figure 6. Time for storing all words in memory

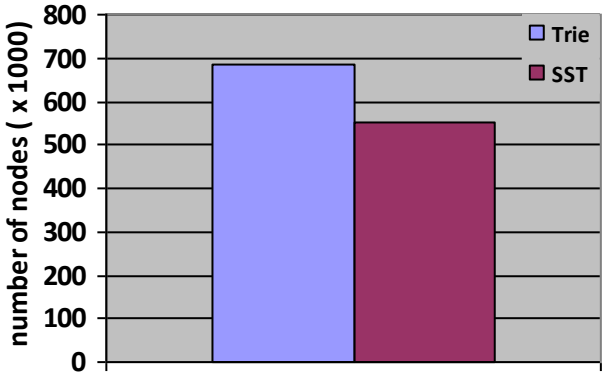

Figure 7. Number of nodes created and maintained in each structure

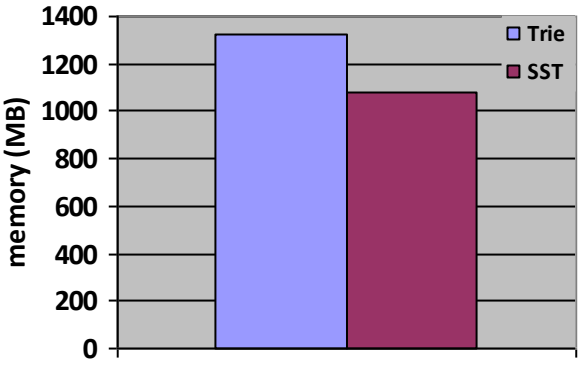

Figure 8. Memory usage to contain all words

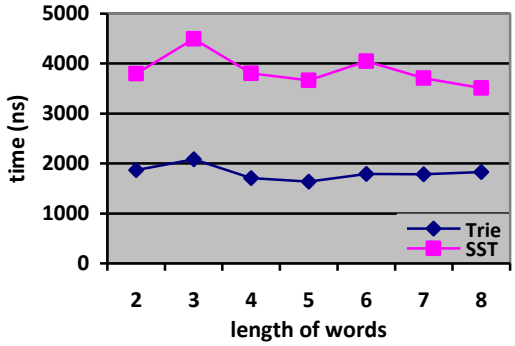

Figure 9. Time for searching words on 2-8 letter words

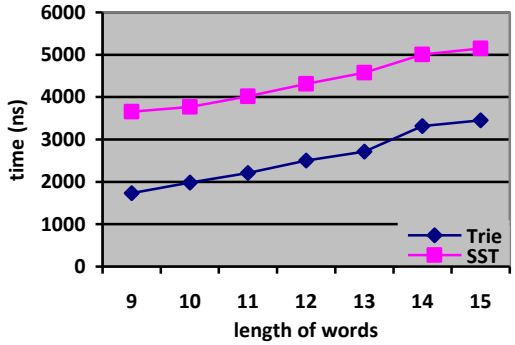

Figure 10. Time for searching words on 9-15 letter words

V. CONCLUSION

This paper studied the problem of dictionary construction that can notify information from a collection of words. Then, an efficient suffix-sharing trie structure (also called *SST*) is introduced to collects all words of languages in memory. *SST* is not only share prefixes as the normal property of tree and trie structure but it can also share suffix. This can help users to identify words that have a suffix as a component and can also reduce memory usage to hold all words comparing with the original trie structure. Experimental results showed that the proposed approach can efficiently maintain dictionary in both runtime and memory consumption. *SST* can alleviate *18.4%* of memory usage and has nearly the same performance on searching as the original trie structure.

In the future, we aim to extract more information from a collection of words which may help users or machine to better understand context. We also try to improve our algorithm to be quickly for storing vocabulary in memory.

ACKNOWLEDGMENT

This work is supported by Faculty of Informatics, Burapha University, Thailand.

REFERENCES

[1] H. Somers, "Review Article: Example-based Machine Translation", in journal of Machine Translation, 1999, pp.113–157.

[2] U. Hahn, and I. Mani, "The challenges of automatic summarization", in journal of Computer, vol. 33, no. 11, 2000, pp.29–36.

[3] T. Liao, Z. Liu, and X. Wang, "Research and Implementation on Event-Based Method for Automatic Summarization", in proceedings of The Eighth International Conference on Bio-Inspired Computing: Theories and Applications, vol. 212, 2013, pp. 103–111.

[4] M. Pesaresi, and J. A. Benediktsson, "A new approach for the morphological segmentation of high-resolution satellite imagery", in IEEE Transactions on Geoscience and Remote Sensing, vol. 39, no. 2, 2001, pp. 309–320.

[5] T. Prill, K. Schladitz, D. Jeulin, M. Faessel, and C. Wieser, "Morphological segmentation of FIB-SEM data of highly porous media", in journal of Microscopy, 2013.

[6] E. F. Tjong Kim Sang, and F. D. Meulder, "Introduction to the CoNLL-2003 shared task: language-independent named entity recognition", in proceedings of the 7th conference on Natural language learning at HLT-NAACL, vol. 4, 2003, 142–147},

[7] A. Golynski, J. Munro, and S. Rao, "Rank/select operations on large alphabets: a tool for text indexing", in proceedings of the 17th annual ACM-SIAM symposium on Discrete algorithm, 2006, pp. 368–373.

[8] A. Appel, and G. Jacodson, "The world's fastest scrabble program", Communications of the ACM, vol. 31, no. 5, 1988.

[9] J. Bently, and R. Sedgewick, "Fast algorithms for sorting and searching strings", in proceedings of SODA'97, 1997.

[10] R. Agrawal, and R. Srikant, "Fast algorithms for mining association rules", in proceedings of the 20th international conference on Very Large Databases, 1994.

[11] A. Amir, R. Feldman, and Reuven Kashi, "A new and versatile method for association generation", in journal of Information Systems, vol. 22, 1997, pp. 333–347,

[12] F. Bodon, "A trie-based APRIORI implementation for mining frequent item sequences", in proceedings of the 1st international workshop on open source data mining: frequent pattern mining implementations (OSDM '05), 2005, pp. 56–65.

[13] S. Sahni, and K. S. Kim, "Efficient construction of multibit tries for IP lookup", in IEEE/ACM Trans. Netw, vol. 11, no. 4, 2003, pp. 650-662.

[14] W. Jiang, and V. K. Prasanna, "A memory-balanced linear pipeline architecture for trie-based ip lookup", in proceedings of HotI '07, 2007.

[15] C. Silverstein, "A practical perfect hashing algorithm", DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 59, 2002, pp. 23–47.

[16] R. J. Enbody, and H. C. Du, "Dynamic hashing schemes", ACM *Computing Surveys*, vol. 20, no. 2, 1998.

[17] M.V. Ramakrishna, and J. Zobel, "Performance in practice of string hashing functions", in proceedings of international conference on Database Systems for Advanced Applications, 1997, pp. 215–223.

[18] D. Knuth, "The art of computer programming", in a book of Sorting and Searching, vol. 3, Addison-Wesley, 1973.